by DJ Delorie

Construct a USB GPIO Pod (Part 1)

No Parallel Port, No Problem

Unless you've been in a cave for the past decade, you know that the parallel port has been eclipsed by the USB port. In this article, DJ describes how to design a general-purpose input output (GPIO) pod that can plug into your USB port. Now you will have the flexibility to define the I/O you need.

I have been fiddling with electronics since before the arrival of the PC. In the early 1980s, it was easy to add custom electronics to a computer. The S100 and ISA busses were easy enough to interface with; but for ease of hook-up, nothing beat the parallel port with 13 output pins and five input pins, which was directly addressable in software and TTL-compatible. These days, it's getting hard to find parallel ports on computers, especially on laptops. Even when you can find one, it is not always compatible with today's 3.3-V logic. The USB port has taken the top spot in popularity.

In the first part of this two-part article series, I'll show you how to make a general-purpose input output (GPIO) pod that plugs into your USB port (see Photo 1). Next month, I'll describe how you can use it to program a CPLD.

USB OVERVIEW

There have been plenty of articles about USB, so I'll just cover the basics. USB is a high-speed serial port with a power feed and only four wires: two for data, one for 5 V, and one for ground. A packet protocol is used to communicate between the host and the devices, much like PPP or Ethernet is packet-based. In addition, the USB host can provide up to 500 mA on the 5-V line, as long as the device has requested it and the host has agreed to provide it. This makes USB a much more flexible connection, but more complex to interface with. Fortunately, there are interface chips that hide this complexity and offer a standardized interface.

The pod's first major component is a Future Technology Devices International FT232R USB-to-UART interface chip. On the USB side, it handles the physical and logical interface to the USB serial lines and manages power to the remainder of the device, if needed. On the UART side, it provides a standard asynchronous serial port, with full flow control and five additional user-assignable I/O pins (CBUS0 through CBUS4).

FTDI provides two libraries that can be used to interface to these chips. The first treats the chips strictly as standard UARTs. The second provides access to the chip's full capabilities. The libraries are available for both Windows and Linux, but the UART library usually is not needed. Both operating systems recognize the chips and automatically load UART drivers for them when a USB device is first plugged in.

R8C OVERVIEW

The second major component in the GPIO pod is a Renesas Technology R8C microcontroller. It is a 16-bit microcontroller that can run at either 3.3 or 5 V, at up to 20 MHz, with built-in flash memory, RAM, and a wide variety of built-in peripherals. While the FT232R chip has user-controllable I/O pins, using a microcontroller gives you much more

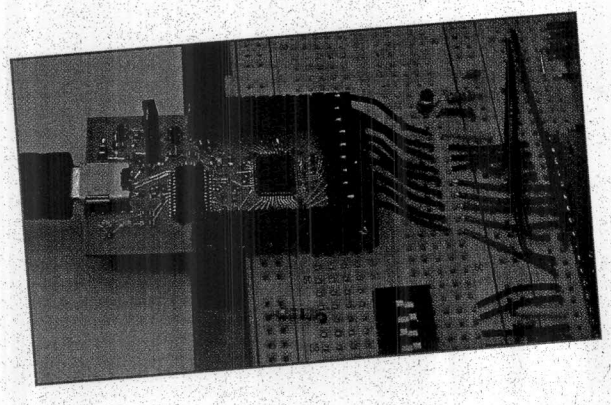

Photo 1—The pod can be used to interface your host PC to a project. Here it provides input data to a CPLD project.

speed and flexibility. For example, the R8C has an ADC, so you can use it for measuring analog signals. You can also download software into the R8C to turn it into a "smart" I/O pod, which will enable you to optimize the USB serial datastream. Next month, I'll explain how I downloaded an entire JTAG programming application into the pod and sent the compressed CPLD bitstream to it through the serial port.

There are a couple of reasons why I chose the R8C. You may remember the 2005 Renesas M16C Design Contest. The R8C is the M16C's little brother, available in packages with as few as 20 pins. It has two characteristics that make it ideal for this project. One, it can run off either 5 or 3.3 V, enabling it to interface to a wider range of projects. Two, you can program it using nothing more than a serial port and two GPIO signals, one for nRESET and one for MODE.[1]

I used DTR to control nRESET, CBUS2 for MODE, and feed the UART lines into the R8C's serial port. By doing this, you can use the USB connection to program the R8C and communicate with that program, so no additional programming connectors are needed! How does this work? Well, the R8C (and M16C and M32C, as well) has two flash memory blocks. One is for the user program, which is the flash memory that is described in the chip's hardware manual. The second is a small flash memory block that includes a simple bootstrapping program. When the chip comes out of reset (the nRESET line goes high), it samples the MODE pin to determine from which flash memory block to boot. Normally, MODE is pulled high, and resetting the chip runs your program. If MODE is low, the bootstrap program runs instead, enabling you to download a new program.

The specific R8C chip used here is the R8C/20 chip in a 48-pin TQFP. It has six byte-wide I/O ports. Four ports (Ports P0, P1, P2, and P6) expose all 8 bits to pins on the chip, while two (P3 and P4) expose only 6 of the 8 due to the limited number of pins. Port P6 is used for the seven UART signals (DTR goes to nRESET), leaving 1 bit to control an on-board LED. Ports P0, P1, P2, and P3 are brought out to standard 0.1" headers. Two of the P4 pins are used to make up for the missing P3 bits. Two of the remaining P4 pins are used for the ADC reference voltage and the clock.

I chose this chip because it has sufficient I/O pins to do nearly anything I can imagine. P0 has eight ADC inputs. P2 has full three-phase motor control capabilities. P1 offers six interrupt inputs, four more ADC, and a dual-mode (synchronous and asynchronous) serial port. P3 has an SPI/I²C interface and two timer outputs. As you can see, there are a lot of peripherals for project-specific interfacing and intelligence.

THE USB GPIO POD

Given what I have covered so far, the pod almost designs itself. The FT232R chip lets the host talk to the R8C, either to program it or to communicate with it. The R8C controls

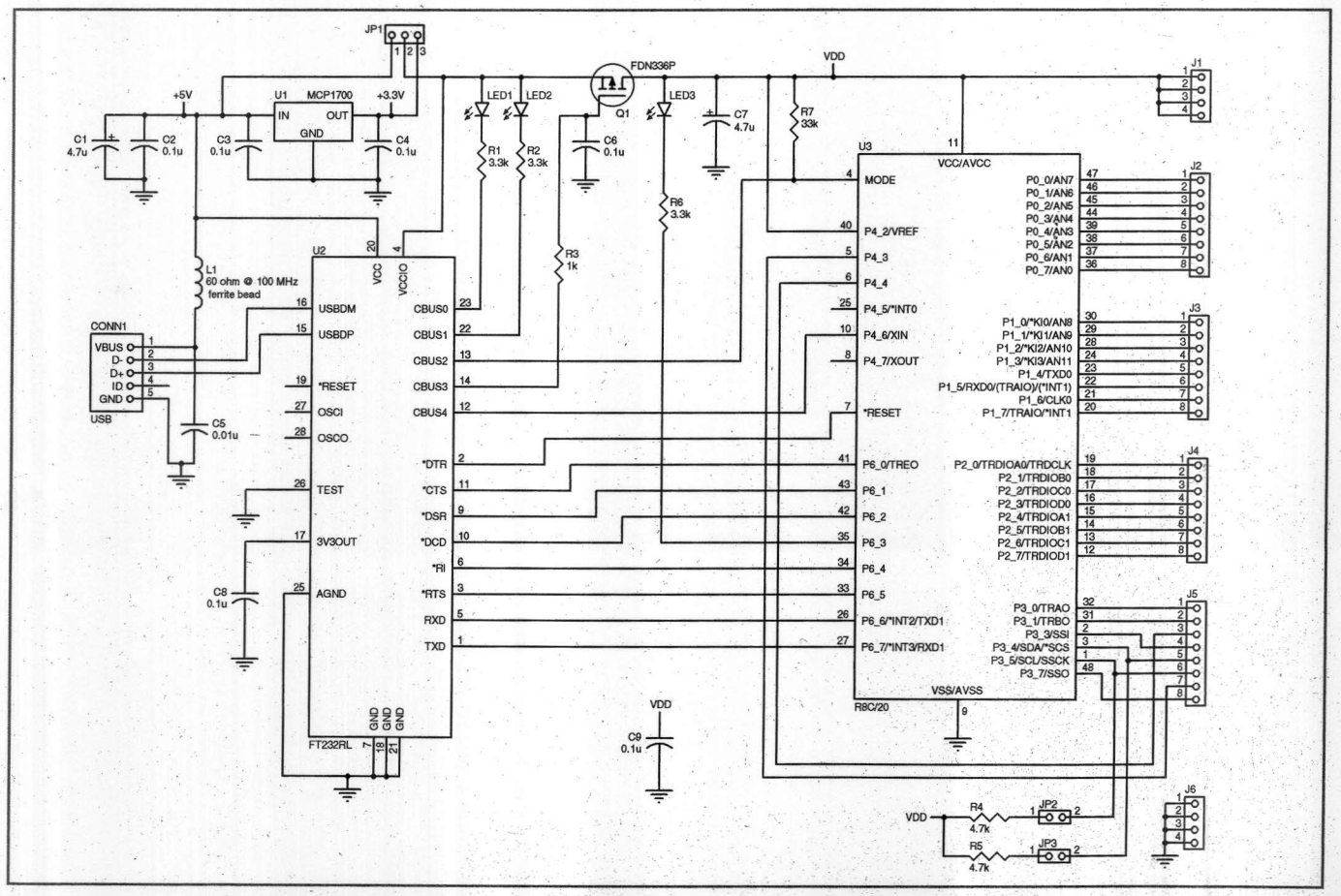

Figure 1—Note the simplicity of the circuit. Most connections go between the R8C (U3) and either the USB chip (U2) or the headers. The rest of the circuitry is mostly for power control. There are also LEDs and jumpers for I²C pull-ups.

the various I/O ports on the headers according to its programming. The headers include power and ground, so the pod can power your project and talk to it. The headers accept standard 22-gauge solid wire, just like a solderless breadboard. Or you can make modules that plug onto the headers like a daughter board to add additional circuitry or change the connector pinouts depending on the project (see Photo 1).

I chose a small board—just over 2" × 1"—so it could be used more like a "pod" and less like a "board." It's small enough to be considered part of the USB cable. One end of the board is the USB connector and power management circuit, followed by the FT232R chip. The other end is the R8C chip surrounded by headers. The wiring also follows this flow: the USB connector is wired to the FT232R and power circuits, the FT232R talks to the power circuits and the R8C, and the R8C connects to the headers (see Figure 1).

Listing 1—This is a simple pod application that toggles output pins.

```
#include "bsp.h"

main()
{
   unsigned char j;

   /* Configure port 1 for all GPIO output.  */
   pd1.b = 0xff;

   while (1)
   {
      j ++;
     /* Write the new value out to port 1.  */
      p1.b = j;
      wait_ms (100);
   }
}
```

The USB connector provides 5-V of power to the device. However, there are some rules about using this power. The device is limited to 500 µA in "suspend" mode and 100 mA during USB negotiation. If the device and host agree, the device may use up to 500 mA, although the pod is rated for only up to 250 mA. The pod has a 3.3-V regulator U1 and a jumper to choose between 3.3- or 5-V operation. The jumper controls power to the FT323R's UART pins, the R8C chip, and the headers. The jumper's output is switched with a P-MOSFET controlled by the FT232R, so the FT232R can keep the R8C powered off until it and the host agree on power requirements.

The FT232R chip manages the USB connection. When you plug in the USB cable, it negotiates the data rate (up to 12 Mbps) and power needs with the host. The FT232R has an internal EEPROM that contains configuration information, such as device identification and power needs. I programmed mine to be named "usb-r8c," and I asked for 250 mA, using utility software available at FTDI's web site. Once the power is negotiated, CBUS3 is pulled low to activate Q1, supplying power to the R8C chip and headers.

The connections between the FT232R and the R8C are fairly straightforward. The UART signals go to P6, except DTR, which is used for nRESET. I took care to match the Tx and Rx pins so the R8C's internal UART can be used to communicate with the FT232R. CBUS2 is used to control the MODE pin. CBUS4 provides a 12-MHz clock from the FT232R to the R8C. An external crystal is not required. Because the R8C runs at up to 20 MHz, you can use a separate crystal to increase its performance. Twelve megahertz is just the fastest clock available from the FT232R that doesn't exceed the R8C's limits. You could also use an 18.432-MHz crystal if you need accurate UART signals, because 18.432 MHz is 160× faster than 115,200 bps, and thus is the fastest clock under 20 MHz that results in an exact divisor in the UART clock control register. The UART needs a clock 16× faster than the data rate, and 115,200 × 16 × 11 is 20.275 MHz. However, running off the FT232R's clock enables you to coordinate data rate divisors between the FT232R and the R8C, allowing fast communication with the host. Finally, you can use the R8C/20's internal oscillator, which is the equivalent of a 20-MHz crystal but is not as precise.

On the other side of the R8C, I just connected each pin to one of the header pins. Because P3 is missing 2 bits, two of P4's bits fill in, providing 32 I/O pins in four groups of eight. Additionally, four ground and four power pins are provided in the headers. Because I use I²C a lot, I included two I²C pull-up resistors and jumpers to enable them. There is also an on-pod LED connected to pin P6.3, which can be used for diagnostics. The pod's host-side serial port driver in my software uses this LED to reflect the flow control status between the host and the pod.

PROGRAMMING & OPERATION

The pod can be used in one of two modes: Programming or Operation. In both modes, the R8C is reset by temporarily dropping DTR. Because the Linux built-in FTDI drivers do this automatically when the data rate is set to zero, it can be done easily whether you use the built-in drivers or the FTDI library. Note that the FT232R has programmable polarity for the serial port signals. I programmed the DTR line to be active high, so the chip is reset when DTR is dropped and runs when DTR is asserted. DTR is always driven, so no pull-up resistor is needed.

In Programming mode, the R8C MODE line is held low during reset by using the FTDI library to set CBUS2 low, putting the chip in Bootstrap mode. Normally, the CBUS2 pin is tristated, so a pull-up forces the chip into normal running mode when the FT232R is initially plugged in. Thus, only the programming software needs to know about the extra functionality. Once in Programming mode, the programming utility talks to the bootstrap firmware over the standard serial lines to download the new software into the R8C flash memory. It can then raise CBUS2 and reset the R8C to normal running mode.

Once the R8C is programmed, a host-side utility talks to the R8C using either the FTDI library or, more likely, a standard serial port emulation. In Linux, for example, plugging in the pod causes a device like /dev/ttyUSB2 to be created, which can be accessed like any

Listing 2—This is a more complex pod application, showing how to interact with the host PC and the on-chip peripherals.

```
main()
{
  int period, width, channel;

  setup_hardware ();

  while (1)
    {
      switch (tty_readc ())
    {
    case 'o': /* output bits */
       p2.b = tty_readc ();
       break;
    case 'i': /* input bits */
       tty_putc (p0.b);
       break;
    case 'a': /* sample ADC */
       channel = (tty_readc () & 0x03) + 8;
       tty_putc (read_adc (channel));
       break;
    case 't': /* timer */
       period = tty_readc ();
       trbpr = period / 2;
       trbsc = (period + 1) / 2;
       break;
    case 'T': /* timer, with width control */
       period = tty_readc ();
       width = tty_readc ();
       trbpr = width;
       trbsc = period - width;
       break;
    }
    }
}
```

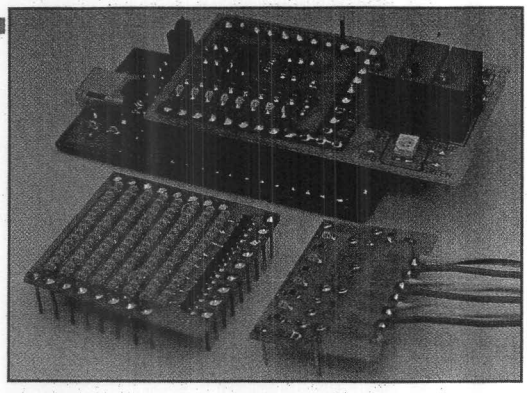

Photo 2—In addition to plugging wires into the headers, you can create a variety of modules that plug into them. Here you see the pod with my LED workbench module, which is used to characterize RGB LEDs. A raster display (lower left) and a JTAG adapter (lower right) are also shown.

other serial port. The actual protocol to use depends on the firmware programmed into the R8C, allowing great flexibility in customizing it for whatever purposes the pod is used for.

I did what everyone does first with circuits like this. I made a blinky light. But before I discuss my application, let me take a moment to discuss what goes on behind the scenes. Each piece of hardware needs a board support package (BSP), which includes all the ancillary support routines that are specific to your hardware (e.g., how to set up the stack, initialize memory, program timers, and more). All of the support files needed for this board are posted on the *Circuit Cellar* FTP site. I won't cover them further in this article, but assume that each program I cover here needs to be linked with the BSP files. My first pod program is in Listing 1.

First, I had to compile this from source format into a binary format that my programming tool would understand. I used the freely available GNU toolchain (specifically, GNU's GCC and Binutils, and Red Hat's Newlib) because it runs on Linux, my platform of choice. Instructions for obtaining and building the GNU toolchain for the M32C family posted on the *Circuit Cellar* FTP site. Each tool's name is prefixed with "m32c-elf-" to indicate that it's for cross-compiling to the M32C family and produces ELF format binaries, which is the most popular embedded file format these days. The Makefile posted on the *Circuit Cellar* FTP site uses m32c-elf-gcc to build the blinky.c into blinky.elf.

The programming tool is called uflash (for USB-based FLASHing tool). Running it downloads blinky.elf to the pod:

```
$ sudo ../uflash/uflash blinky.elf
```

USB devices are not normally writable by users, so the `sudo` command gives me permission to access it. The uflash tool talks to the bootloader in the R8C chip to program it. When finished, it resets the chip back into normal mode, and my blinky program runs. Plug a few LEDs and current-limiting resistors into port 1's headers and you've got blinky lights!

"Big deal," you say.

Well, I did something more interesting and made the pod interactive. I created a pod program that offers eight output signals on port 2, eight digital input signals on port 0, four analog inputs on port 1, and a clock output on port 3. All of these are controlled by the host PC. The more complex pod program is in Listing 2.

The call to `setup_hardware()` configures all the I/O ports and peripherals. It's handy to have a copy of the "R8C/20 Group, R8C 21 Group Hardware Manual," on hand, because there are many complex peripherals. For example, in this case, Timer B drives pin P3.2, but the pin can't be driven in standard timer mode, so configure it for function generator mode. That also lets you control both the period and the duty cycle. These details, along with the myriad control registers that configure them, are all detailed in the manual.

All pod programs need some sort of "loop forever" in `main()`; because there's nothing to return to. In this case, the peripherals are doing all the work, so my main loop is used to wait for data from the host. The protocol is simple. A command byte is followed by zero or more data bytes, and it may cause zero or more data bytes to be sent back to the host. The `o` command is followed by a byte, which is then sent

to port 2. The I command causes the pod to read port 0 and send the byte back to the host. The a command is followed by a byte that selects one of the four ADC inputs on port 1, reads the ADC value as an 8-bit value, and sends that back to the host. The t and T commands are followed by 1 or 2 bytes that get programmed into Timer B.

PODS AND MODULES

In addition to connecting wires directly to the headers, I have three modules I made for my pod (see Photo 2). One module is an "LED Workbench" that lets me experiment with red and RGB LEDs. One has an 8 × 8 grid of red/green cells. The third, which I'll cover in the second part of this article series, is a JTAG adapter. Each module's design starts with a schematic for the connectors and a PCB layout that places them to mate with the pod. Thus, I can quickly design a new pod module as needed.

The LED workbench includes eight red LEDs controlled by port 2 and an RGB LED controlled by port 1. The RGB LED has adjustable limiting resistors to help you determine the ideal resistors for a good white balance. The ADC inputs monitor the LED voltage drops as well as the voltage across the resistors. Three 10-Ω resistors and three op-amps let the pod monitor the current through each LED. This lets me test and characterize various RGB LED offerings. The pod software monitors the ADCs, disables the LEDs if excessive current is detected, and scales all the values to sensible values. It includes a command-line interface so any terminal emulator program that can talk over the serial port is sufficient to talk to the pod. I can also experiment with various LED control algorithms with the eight red LEDs.

Another module is a raster of red and green LEDs, creating an 8 × 8 grid of red/green pairs. (Yes, that's 128 LEDS, along with eight transistors and 16 resistors, in a 1" square space.) This forms a 64-pixel micro display, with port 1 controlling the rows, port 2 controlling the green LEDs in each column, and port 0 controlling the red LEDs in each column. I use this as a scrolling marquee display for monitoring data from my PC. The pod software maintains an 8 × 8 array of what should be displayed, 1 byte per LED. The timer interrupt iterates through each row, enabling a single bit in port 1 and outputting appropriate data to ports 0 and 2, according to the desired intensity of each LED in that row. Meanwhile, data from the host PC is used to fill in new columns of data. When a column has been received, the data for the display is shifted one column over, providing a scrolling marquee style display, with data provided by the host but LED control handled by the pod.

In addition to these, you could use the pod to interface with SPI, I²C, 1-Wire, or serial devices. The R8C's internal ADC could be paired with a D/A chip or two. The pod has a fully functional and programmable computer chip on it, so the possibilities are endless!

YOUR TURN

If you need to turn a bit on and off, this isn't the fastest way to do it. But

if you can offload some of the logic to the pod and do a lot more than just turning a bit on and off, this project gives you the flexibility to define the I/O functionality you need.

Next month, I'll show you how to offload an entire application to the pod and feed only its data files over the USB link. Stay tuned.

DJ Delorie (dj@delorie.com), who has been designing electronic circuits since high school, earned an ECE degree at Clarkson University. After holding jobs designing PC motherboards and network management software, he now writes embedded development tools for Red Hat. DJ is also the creator of DJGPP and one of the contributors to the gEDA project.

PROJECT FILES

To download code, go to ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2009/225.

REFERENCE

[1] D. Delorie, "Easy R8C/M16C/M32C Flash Programming," 2008, http://people.redhat.com/dj/m32c/flash-guide.pdf.

RESOURCES

D. Delorie, "USB GPIO Project Page," www.delorie.com/electronics/usb-gpio.

GNU Operating System, Free Software Foundation, Inc., www.gnu.org.

Renesas Technology Corp., "R8C/20 Group, R8C 21 Group Hardware Manual," 2008, http://documentation.renesas.com/eng/products/mpumcu/rej09b0250_r8c2021hm.pdf.

SOURCES

GNU Binutils
Free Software Foundation, Inc. | http://sourceware.org/binutils

FT232R USB UART IC and MProg 3.5 EEPROM Programming utility
Future Technology Devices International | www.ftdichip.com/Products/FT232R.htm

Newlib
Jeff Johnston | http://sourceware.org/newlib

R8C Microcontroller
Renesas Technology Corp. | www.renesas.com/en/r8ctiny

GCC Operating system
The GNU Compiler Collection | http://gcc.gnu.org